Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

PAI 3 - BYODSEC-Bring your own device seguro para una universidad pública usando Road Warrior VPN SSL

Grado en Ingeniería Informática – Ingeniería del Software

Seguridad en Sistemas Informáticos e Internet

Curso 2023 – 2024

Participantes:
Juan Luis Ruano Muriedas

José Joaquin Rojas Romero

Antonio José Suárez García

Índice

Resumen ejecutivo

En este proyecto implementaremos un **Socket** de tipo Secure **Sockets Layer (SSL)** para cumplir con el requisito de una comunicación cliente-servidor confidencial, autenticada e íntegra.

Para enfrentar estos desafíos, se ha desarrollado un **código en Java** que utiliza **una keyStore** con algoritmo simétrico **RSA** generada a través de la utilidad **keytool**, la cual es compartida de manera segura entre el cliente y el servidor a través de una **memoria USB**.

Para evitar posibles ataques en los **cipher suites** empleados en el proceso de handshake, se ha hardcodeado **el mejor cipher suite posible** entre los pocos que soportan **TLS1.3** (**TLS_AES_256_GCM_SHA384**) para garantizar la máxima seguridad en la información que se transmite y recibe.

Finalmente se ha realizado un **script de prueba de carga** para garantizar, que al menos, **300 usuarios** pueden beneficiarse de ese sistema **sin congestión o errores**.

Se ha diseñado el código para ser compilado y ejecutado sin tener que emplear o conocer largos comandos o conocimientos en Java, aportando rapidez y ahorrando posibles errores humanos, pero también puede ser ejecutado a través de comandos completos por terminal tal y como se mostraba en la documentación, aportando toda la flexibilidad que nuestras capacidades nos han permitido.

Road Warrior VPN SSL

Estructura del código del cliente

```java
public class MsgSSLClientSocket {

    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        try {

            SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
            SSLSocket socket = (SSLSocket) factory.createSocket("localhost", 3343);

            //TrustStore setting
            System.setProperty("javax.net.ssl.trustStore", "/home/tric0/Escritorio/PAI3/PAI_3_code/k1.jks");
            System.setProperty("javax.net.ssl.trustStorePassword", "654321");
            //Accepted Cipher suites
            System.setProperty("javax.net.ssl.cipherSuites", "TLS_AES_256_GCM_SHA384");
```

Figura 1. Configuración de la Trustore del cliente

Para la estructura del lado del cliente se ha configurado la ***Trustore*** con la anteriormente recibida Java ***Keystore*** a través de un USB y estas líneas son de configuración que preceden a las de conexión del socket para la configuración de la conexión. Lo más importante de la configuración es la selección de *cipher suites* que se usarán en la conexión el cual hemos elegido y configurado el **TLS_AES_256_GCM_SHA384** para asegurar la confidencialidad. El restante del código se trata del socket en Java para la conexión con el servidor.

Estructura del código del servidor

```java

    /**
     * @param args
     * @throws IOException
     * @throws InterruptedException
     */
    public static void main(String[] args) throws IOException, InterruptedException {

        try {
            SSLServerSocketFactory factory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
            SSLServerSocket serverSocket = (SSLServerSocket) factory.createServerSocket(3343);


            //KeyStore setting
            System.setProperty("javax.net.ssl.keyStore", "/home/tric0/Escritorio/PAI3/PAI_3_code/k1.jks");
            System.setProperty("javax.net.ssl.keyStorePassword", "654321");
            //Accepted Cipher suites
            System.setProperty("javax.net.ssl.cipherSuites", "TLS_AES_256_GCM_SHA384");

```

Figura 2. Configuración de la keyStore del servidor.

Al código que se nos entregó para el servidor solamente se han añadido tres líneas de código(líneas 23, 25 y 27) para indicar la configuración que tendrá el servidor. En las líneas 24 y 25 se indica la keyStore y su contraseña, para así no tener que hacerlo en la consola de comandos al iniciar el servidor, facilitando así su uso. Por último, en la línea 27 se indica el *cipher suite* que se usará, en este caso **TLS_AES_256_GCM_SHA384.**

Comprobación de la confidencialidad del tráfico

Para comprobar la confidencialidad del tráfico se ha empleado la herramienta Wireshark donde se han podido comprobar las tramas TLS intercambiadas durante el proceso de handshake y comunicación cliente-servidor.

En la trama Client Hello se puede comprobar un listado de cipher suites posibles y en la trama Server Hello se selecciona una de ellas, en nuestro caso le indicamos que siempre emplee **TLS_AES_256_GCM_SHA384** ya que es de las mejores opciones y compatibles con TLS1.3.

Pruebas

Todo el código ha sido probado y testeado, en primera instancia, de manera informal observando los resultados en la terminal tras ejecutar los dos scripts. Para comprobar la carga de 300 usuarios, se ha elaborado un script que simula mediante hilos a los 300 usuarios y realiza una conexión con el servidor donde no se pregunta a través de ninguna ventana la contraseña o clave de acceso que se pedía de manera normal a través de una ventana JOptionPane sino que dicha palabra clave ya se encuentra hardcodeada y se introduce automáticamente en cada conexión.

```
11  public class MsgSSLClientLoadTest {
12
13      public static void main(String[] args) {
14          int numClients = 300;
15
16          // Crear y ejecutar los 300 hilos
17          for (int i = 0; i < numClients; i++) {
18              new ClientThread().start();
19          }
20      }
```

Figura 3. Captura del bucle con 300 iteraciones modificable a las iteraciones deseadas. En cada llamada se inicia un hilo nuevo que simula a un cliente independiente.

```java
private static class ClientThread extends Thread {
    @Override
    public void run() {
        try {
            // Configurar el cliente SSL
            SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
            SSLSocket socket = (SSLSocket) factory.createSocket("localhost", 3343);

           // create BufferedReader for reading server response
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            // create PrintWriter for sending login to server
            PrintWriter output = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            // prompt user for user name
            //String msg = "Hola";
            String msg = "Hola";
            // send user name to server
            output.println(msg);

            output.flush();

            // read response from server
            String response = input.readLine();

            // display response to user
            System.out.println("OK");
            //JOptionPane.showMessageDialog(null, response);


            // clean up streams and Socket
            output.close();
            input.close();
            socket.close();

        }

        catch (IOException e) {
            e.printStackTrace();
        }
        // exit application

        finally {
            System.exit(0);
        }

```

Figura 4. Captura del método principal ejecutado por cada hilo en la captura justo anterior..

Como instalar el VPN

El código descrito anteriormente está adjunto a este documento o lo puede encontrar en nuestro repositorio de *Github*: https://github.com/tric0ma/CAI-1/tree/main/pai-3.

GUIA de instalación en Linux y pasos para este PAI 3

- Para comprobar si tenemos o no Java instalado:
  java -version
  which java

- Instalar última versión openjdk:
  sudo pacman -S jdk-openjdk

- Comprobamos que está instalado correctamente y la versión del JDK:
java -version

- Creación de un keystore para la compartición de claves SSL/TLS de ejemplo:
keytool -genkeypair -keystore k1.jks -alias SSLCertificate -keyalg RSA
    * Nos pedirán datos, en mi caso pongo contraseña: 654321
    * Nombre: SSII
    * Organización: US
    * City: Seville
    * Location: Seville
    * Two letter country: ES
    * Confirmation: si (responder en el idioma de tu país)

- Listamos el keystore generado para comprobar si se ha creado correctamente:
keytool -list -keystore k1.jks

En este punto deberíamos de tener el keystore creado. En nuestro caso hemos compartido el archivo generado k1.jks a través de una memoria USB entre el cliente y el servidor.

- Ahora compilamos las dos clases java para poder usarlas desde la terminal con el comando: **javac *.java**

- Ahora que podemos lanzar las clases desde la terminal, lanzamos el servidor:
java -Djavax.net.ssl.keyStore=k1.jks -Djavax.net.ssl.keyStorePassword=654321 MsgSSLServerSocket

- Desde el lado del cliente ejecutamos el siguiente comando:
java -Djavax.net.ssl.trustStore=k1.jks -Djavax.net.ssl.trustStorePassword=654321 MsgSSLClientSocket
En este punto funciona todo como esperado empleando el mismo host tanto para el cliente como para el servidor como muestra la **Figura 3**.

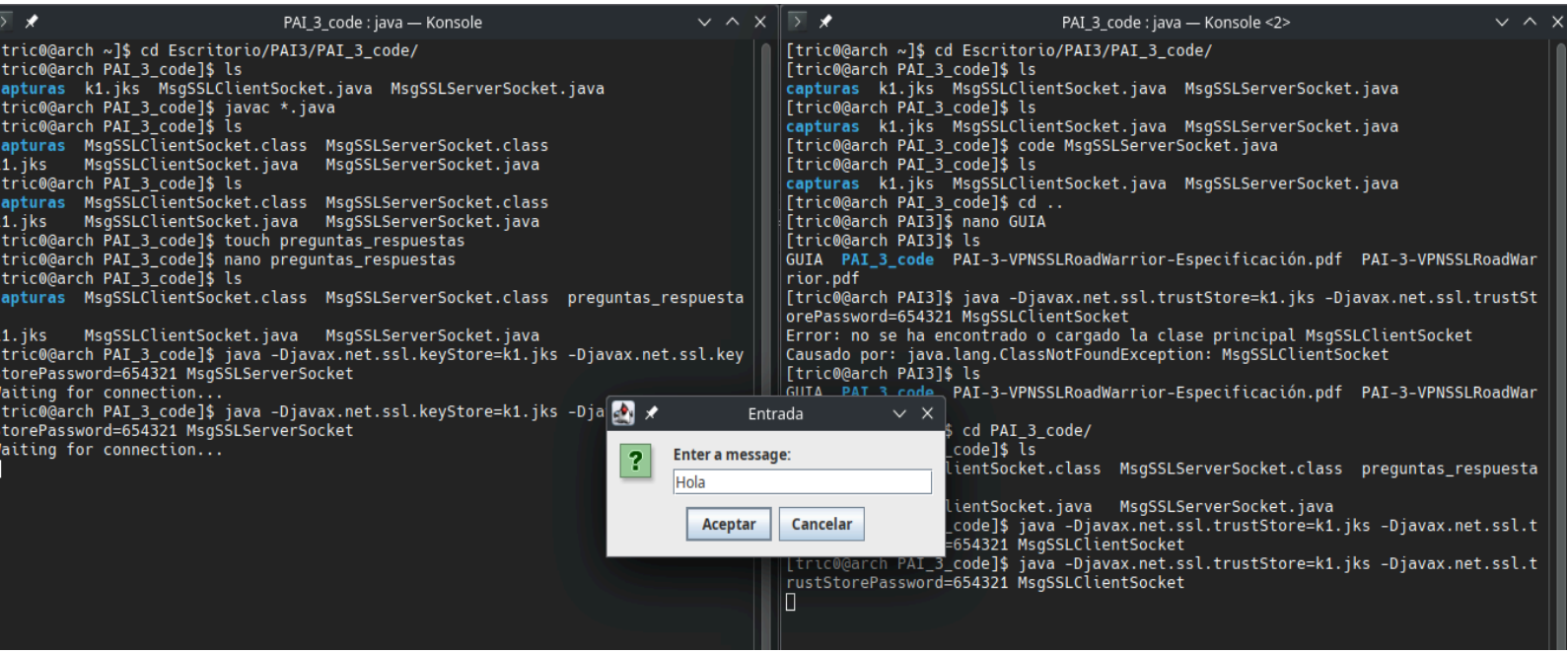

Figura 5. Captura de las pruebas de parte del servidor y cliente en el mismo host, en este caso en un sistema operativo Linux.

Para ejecutar el cliente en un host diferente al servidor, dentro de la misma red, simplemente editamos el archivo MsgSSLCLient.java de manera que la IP que se emplea sea la del host servidor, en vez de 'localhost'.

El host sniffer es una máquina virtual VirtualBox con Ubuntu 20.04.06 en modo promiscuo con Wireshark instalado. Dicha máquina se encuentra activa en el mismo host que el host cliente, es decir, en el host con sistema operativo Windows 10. Puede comprobar en la **Figura 5** el ejemplo de ello.

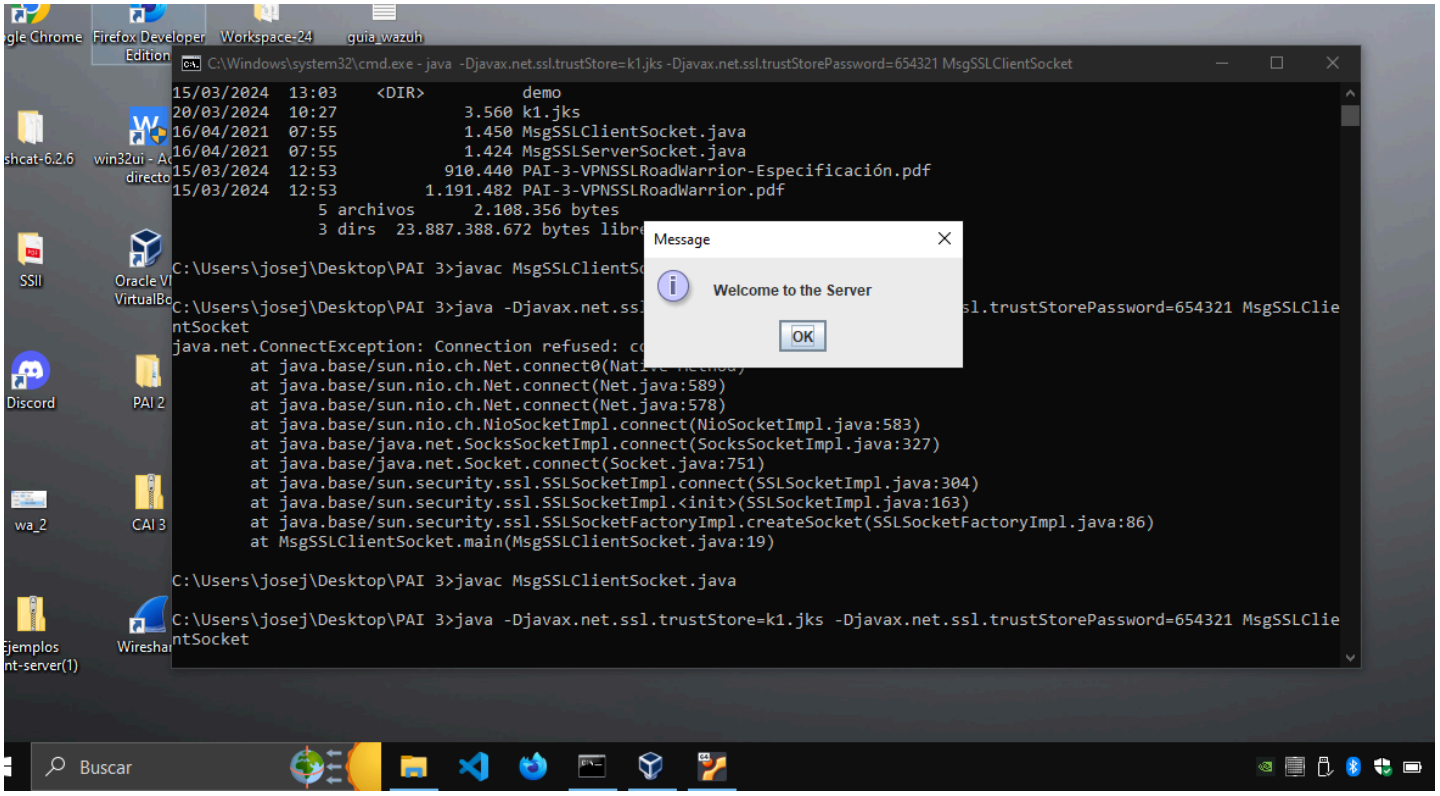

Figura 6. Captura de las pruebas de parte del cliente en un host diferente dentro de la misma red, en este caso un host con sistema operativo Windows 10.

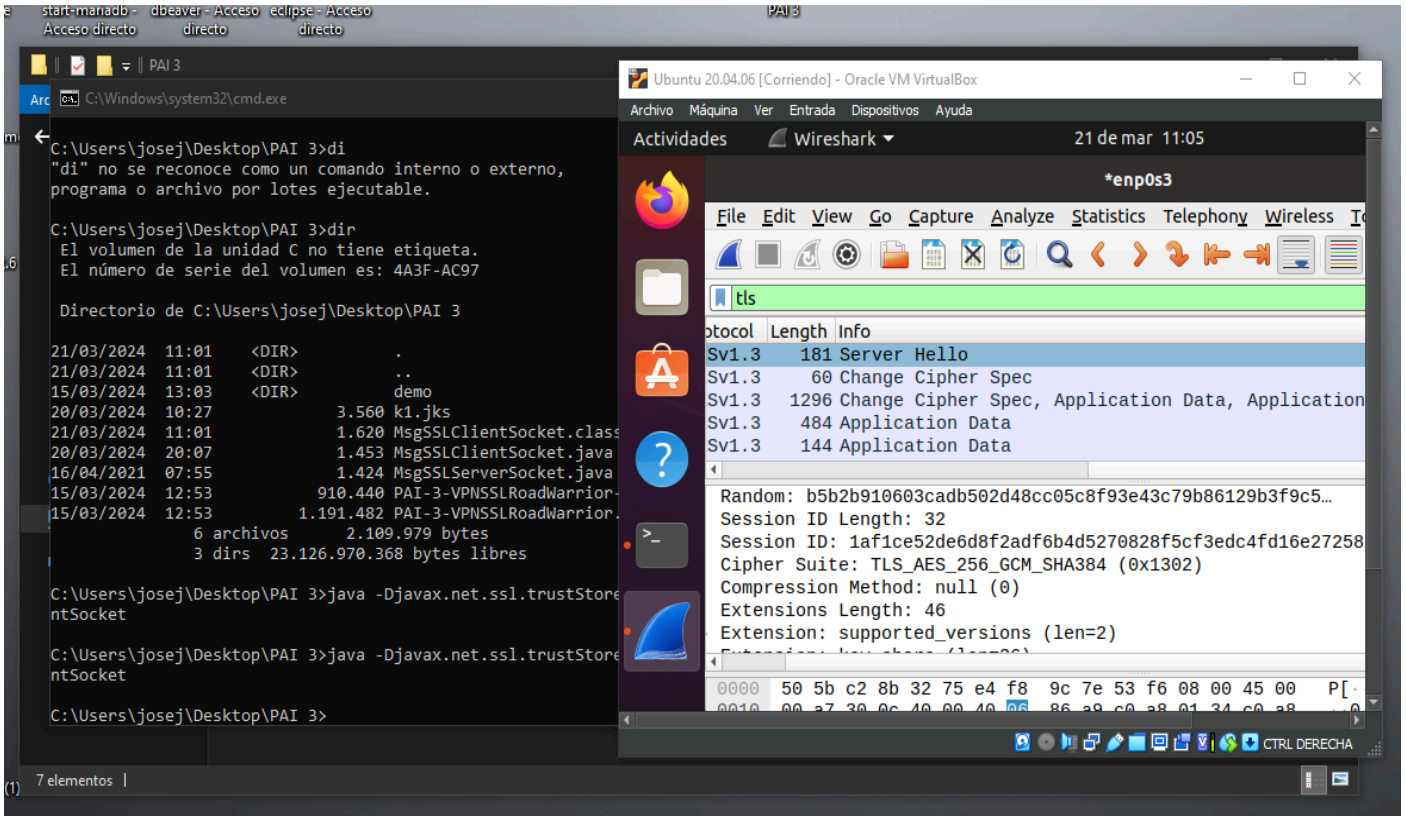

Figura 7. Captura de la máquina virtual con Ubuntu en modo promiscuo ejecutando wireshark a la derecha y mostrando la respuesta del servidor, y a la derecha el cliente de la conexión, ambos en el mismo host pero con distintas IPs.

Conclusiones

Haciendo uso de la tecnología Java Keystore y de TLS 1.3 podemos asegurar la confidencialidad punto a punto de la comunicación incluso a través de redes públicas y a prueba de *sniffing*.